REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
01-02-2012	final	01 March 2009 -01 February 2012

4. TITLE AND SUBTITLE

Software Contracts in a Higher-order World

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-09-1-0110

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Matthias Felleisen

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Northeastern University
College of Computer Science
Boston, MA 02115

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
Suite 325, Room 3112
875 Randolph Street
Arlington, VA 22203-1768

10. SPONSOR/MONITOR'S ACRONYM(S)

AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-OSR-VA-TR-2012-0945

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution A: Approved for public release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The research project investigated foundational models of contracts in a higher-order world of programming. The primary thrust of the work explored the meaning of contracts. We focused on three questions. First, we determined what it means for a first-class function or object to satisfy a contract. Second, we worked out when it is correct for a contract monitoring system to blame a component for violating a contract. We could show that existing contract systems may point to an innocent component and thus send a programmer on a wild goose chase. Third, we established criteria for the completeness of monitoring systems. Using a model, we were able to demonstrate the completeness of one semantics for contract monitors.

We used our primary model to explore designs for the parallel execution of contracts but without reaching a truly satisfactory answer.

The secondary research project explored affine type systems as "protocol contracts" and the use of behavioral contracts to connect an affine code base to libraries from conventional languages. The result of this work is a design for a practical, ML-style programming language with an affine type system and with a contract-based mechanism for integrating existing libraries.

15. SUBJECT TERMS

contracts higher-order programming

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Matthias Felleisen
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU	4	**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U			617-363-2085

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Software Contracts in a Higher-Order World

Matthias Felleisen
matthias@ccs.neu.edu

College of Computer Science
Northeastern University
Boston, MA 02115

February 1, 2012

Abstract

The research project investigated foundational models of contracts in a higher-order world of programming. The primary thrust of the work explored the meaning of contracts. We focused on three questions. First, we determined what it means for a first-class function or object to satisfy a contract. Second, we worked out when it is correct for a contract monitoring system to blame a component for violating a contract. We could show that existing contract systems may point to an innocent component and thus send a programmer on a wild goose chase. Third, we established criteria for the completeness of monitoring systems. Using a model, we were able to demonstrate the completeness of one semantics for contract monitors.

We used our primary model to explore designs for the parallel execution of contracts but without reaching a truly satisfactory answer.

The secondary research project explored affine type systems as "protocol contracts" and the use of behavioral contracts to connect an affine code base to libraries from conventional languages. The result of this work is a design for a practical, ML-style programming language with an affine type system and with a contract-based mechanism for integrating existing libraries.

1 Contracts in a Higher-order World

Over the past 30 years behavioral software contracts have become a popular tool. The purpose of a contract is to restrict a function (or method) signature beyond what the language's type system permits. A run-time system monitors contracts; when violations are discovered they are reported; an attempt is made to assign blame to the guilty party to the contract; and an exception is signaled to repair/prevent further harm. Contracts help programmers with testing, debugging, maintenance tasks, and even dynamic systems control.

In contrast to programming languages, software contracts have been stuck in the 1980s. Over the past 15 years, numerous concepts from higher-order functional languages have found their way into mainstream languages, e.g., C# and Javascript. Most prominently, languages now come with first-class functions, and programmers make extensive use of first-class objects and callbacks. In addition, programming languages no longer have only static classes or modules; Clojure, Fortress, and Scala come with traits, a mechanism for dynamically gluing together classes.

The purpose of this project was to explore software contracts in this higher-order world. We started with an investigation of contracts for first-class program components, say, ML's functorial modules and Squeak's first-class classes. Based on this work and our experience with contracts for first-class functions, we then set out to explore the meaning of contracts and the correctness of contract monitoring systems. Finally, we explored affine type systems and the use of behavioral contracts to connect an affine code base to libraries from conventional languages.

2 Contract Basics

Like business world contracts, a software contract is an agreement between (usually two) parties to live up to certain expectations. From the perspective of one party, a contract establishes *promises* and *obligations*. That is, the party promises to deliver certain services, and it expects its contract partner to deliver certain obligations. Syntactic contracts—also known as types—can be checked at compile time—while behavioral contracts tend to require run-time checks.

Here is a concrete example. A module may export a function `F` from integers to integers. Now suppose the module wishes to add that `F` really promises to return a prime number and that is always to be applied to prime numbers. The type systems of current programming languages cannot check that `F` is always applied to prime numbers and that it always produces one. Hence the module interface contains code to check these additional expectations.

For first-order contracts, it is straightforward to determine when a party violates the agreement. As soon as the "flat" value enters the module, the contract system checks whether it satisfies its obligation. If the value fails this test, the client module—that is, the caller of the function or method—is blamed. Conversely, when the function returns a value, the contract monitoring system checks whether the answer is "as promised." Naturally, if the answer fails the test, the server module—that is, the callee—is blamed.

None of these ideas scale to higher-order programming constructs. Contracts in this world are about entire functions, objects, classes, or modules. For example, if a module exports a numerical differential operator, its type says that it maps functions on the real numbers to the same class of functions. One possible contract may state that the resulting function computes an approximation to the slope of the first one for each argument. By Rice's theorem, it is simply impossible to check such a contract all at once. Worse, even if the contract just samples a small number of points, we are facing the central problems of such contracts: they call unknown methods or functions, and doing so comes with all kinds of problems. Lastly, when systems dynamically compose classes or modules, a contract monitor does not even know the parties to a contract at compile time—meaning it doesn't know whom to blame when something goes wrong.

3 Contracts for First-class Components

With Strickland (PhD, degree expected June 2012), Felleisen explored contracts for languages with first-class modules and first-class classes.

Results: DSL 2009, IFL 2009, DSL 2010

They were able to design contract monitors that track the obligations of contract parties, even in systems where contract parties are loaded at run-time and when obligations are created dynamically. The basic idea is to abstract such components implicitly over their partners and to ensure that the information follows higher-order values as they flow through the system. The complexity of these arrangements naturally raised the questions of when contract monitoring systems are correct.

4 Correct Contract Systems

Dimoulas (PhD, degree expected December 2012) and Felleisen investigated this central question for the entire project period. Their major results greatly clarified the correctness question and exposed problems in practical systems.

Results: TOPLAS 2012 (submitted in 2010), POPL 2011, ESOP 2012, PPDP 2009

The purpose of a contract monitoring system is to catch mistakes. As soon as component in a systems fails to live up to expectations, the overall system should know about the failure so that it can take corrective action. The simplest action is to stop the system and to inform the producer of the failing component about the problem. Doing so should help the producer find the source of the problem. Alternatively, the system maintainer may wish to replace the faulty component with a different one that promises to satisfy the same expectations.

Given this purpose description, it is critical that contract systems do not blame the wrong component. Conversely, the monitoring system must not open "secret" channels between components over which "bad" values can flow from one component to another. Otherwise such "bad" values may eventually break a contract but once again, the monitoring may blame the wrong component. In short, correct blame is economically the central element for a monitoring system.

The work of Dimoulas and Felleisen establishes formal criterion for judging contract monitoring systems for higher-order languages. In addition, they provide a proof method to show that a contract system satisfies this central criterion. Using this framework, they were able to show that two of three existing semantics for contract monitoring systems are wrong. That is, an implementation according to these semantics may blame random "by stander" modules and may thus send the programmer or system maintainer on a wild goose chase. Conversely, the two of

them were able to repair an existing semantics so that it blames only "guilty" modules. Furthermore, their framework suggests a natural refinement of this semantics that implements contracts more efficiently than other systems. They failed, however, in their attempt to produce an effective parallel implementation of contracts. That is, the issue of parallelization of contracts remains an open problem.

5 Practical Affine Types from Contracts

Tov (PhD, defense scheduled: 9 February 2012) and Pucella focused on affine type systems and the use of dynamic contracts to link affine software components into a system of components written in conventional languages.

Results: ESOP 2010, POPL 2011, OOPSLA 2011

An affine type is a syntactic contract on resources such as pointers, file handles, or network connections. With affine types, a programming language can for example check at compile time whether a component hangs to pointers contrary to promises. To build a practical programming language with affine types, we need to connect a kernel language with pragmatic libraries, and we must do so in a way that does not violate the invariants of the affine type system. Tov and Pucella were able to show that Findler and Felleisen's higher-order dynamic contracts can overcome the impedance mismatch.

The result of this work is a design for a practical, ML-style programming language with an affine type system and with a contract-based mechanism for integrating existing libraries.